Group CS162 [Ohhhh NO :]

Task 1: Implement KThread.join()

In the **KThread** class, we will include a boolean **joinCalled**, which will keep track of whether **join()** has already been called on this KThread, as well as a field **KThread**, which will keep track of the thread that called *this* KThread.

Implement Fields:

```
// true: this KThread is already called join()
// false: otherwise
boolean joinCalled

// need to store a thread that called this KThread
KThread joinThread
```

The **join()** method will first check if it has already been called. If it hasn't been, then it will keep track of which thread made this call to **join()**, as well as running *this* thread.

Implement join():

```
join()
      if (joinCalled) {
            throws new UndefinedException();
      }
      disable interrupts
      joinCalled = true
      joinThread = callingThread
      this.run()
      make donation callingThread -> currentThread
      enable interrupts
```

The **finish()** method will ensure that the thread that made the **join()** call now resumes running.

Implement finish():

```
private void finish()
      if (joinCalled)
            nextThread = callingThread;
            nexThread.run();
      else
            // keep original code from finish()
```

Task 2: Implement Condition2 class

The state of **Condition2** includes:

- A lock (Lock conditionLock) is the lock associated with this condition variable

- A waiting queue (ThreadQueue waitingQueue) is the data structure to store thread on sleep.

Impelement for sleep()

```
Condition2::sleep()
    Assert the lock is hold by the currenThread
    conditionLock → Release()
    interruption is disabled
    add currentThread to waitingQueue
    currentThread → Sleep()
    conditionLock → Acquire()
    interruption is enabled
```

Implement for wake()

```
Condtion2::wake()
    Assert the lock is hold by the currentThread
    interruption is disabled
    nextThread ← waitingQueue chooses next thread to run
    if (nextThread != null)
        nextThread → Ready()
    interruption is enabled
```

Implement for wakeAll()

```
Condition2:wakeAll()
    Assert the lock is hold by the currenThread
    Thread nextThread;
    interruption is disabled
    nextThread ← watingQueue choose next thread to run
    while (nexThread != null)
        nextThread → Ready();
        choose nextThread to run from waitingQueue
    enable interruption
```

Task 3: Implement Alarm class

Alarm will have an inner class, `WaitingThread`. The state of `WaitingThread` will include:

- A reference to `KThread (KThread thread)`
- The associated wait time a thread have to wait before waking up (long time)

Also, the Alarm class should include a priority queue (`PriorityQueue waitingThreadsQueue`) to store objects of `WaitingThread` class, and its priority is based on waiting time of each thread.

Implement `waitUntil(time)` method:

```
Alarm::waitUntil(time)
    Interruption is disabled
    waitingThread ← new WaitingThread()
```

```
          Add waitingThread to waitingThreadsQueue
          currentThread → Sleep();
          Interruption is enabled
```

Implement `timeInterruption()` method:

```
     Alarm::timeInterruption()
          AssertTrue (interrupts have already been disabled)
          For waitingThread in the waitingThreadsQueue that have
          exceeded its associated wait time
               waitingThread → Ready()
               watingThreadsQueue → remove(waitingThead)
```

Task 4: Implement Conditional Variable inside Communicator

Communicator imitates the same behavior as **producer-consumer** problem but it has the maximum size of 1 for the bounded buffer, one for *speaker* and one for *listener*.

Let's first look at the high-level design of **Communicator** class. The basic class structure will have 3 main operations and necessary instance variables. These operations are the **constructor**, **speak**, and **listen**. Plus, necessary instance variables are **lock**, **isSpeaker condition**, **isListener condition**, **isWord** boolean, and **word** to store message.

Let's now talk about the nature of each instance variable of *Communicator* class.

`word (String, initialized to null)` to keep the message exchange inside the heap shared by all the thread.

`isWord (boolean, initialized to false)` to notify if there is a word to exchange. Let say a *speaker* has sent a word, **isWord** is set to **true.** If there is a *listener* that is running, **isWord** is set to **false.** In this case, *listener* has received message from *speaker* and both finish executing.

lock (Lock) to provide atomic operation on either *speak* or *listen* method. This is lock also belongs to 2 condition variables **isSpeaker** and **isListener.**

isSpeaker (Condition2) to properly signal one *listener* and put *speakers* to sleep and wake one up upon **isListener** signaling.

isListener (Condition2) to properly signal one *speaker* and put *listeners* to sleep and wake one up upon **isSpeaker** signaling.

Let's now consider the implementation of each method.
Communicator constructor: to initialize instance variables.

```
lock = new Lock();
isSpeaker = new Condition2( lock );
isListener = new Condition2( lock );
isWord = false;
word = null; // optional
```

speak method: to send message.

```
void speak ( int word ) {
   acquire lock;
   while ( count == 1 ) { // if there are other speakers
     isSpeaker.sleep(); // wait for signal from listener
   }
   isWord = true;
   this.word = word;
   isListener.wake(); // signal one listener
   release lock;
 }
```

listen method: to receive message.

```
void listen ( ) {
   acquire lock;
   while (count == 0 ) { // if there is no speaker
     isListener.sleep(); // wait for signal from speaker
   }
   isWord = false;
   isSpeaker.wake(); // signal one speaker
   release lock;
   return this.word;
}
```

Testing / Unit Test

```
* one speaker and one listener: speaker calls first
* one speaker and one listener: listerner calls first
* multiple speakers/listeners: all speakers call first
* multiple speakers/listeners: all listeners call first
* multiple speakers/listeners: mix one another
```

Task 5 Impelement PriorityScheduler class

In this task, we are going to implement the **PriorityScheduler** for nachos. We have know that there are two classes inside **PriorityScheduler.**

a. ThreadState
b. ThreadQueue

In the Lock, Semaphore, and Condition variables, we are using **ThreadQueue** as "waiting queue" for other stuff. So we only need to focus on **ThreadQueue** right now.

```
ThreadQueue waitingQueue = … // initialize for waitingQueue
KThread thread = …// initialize for thread
```

There are two threads, we need to look at :

```
      thread.acquire(waitingQueue);
            public void acquire(KThread thread) {
                  Lib.assertTrue(Machine.interrupt().disabled());
                  getThreadState(thread).acquire(this);
            }
      thread.waitForAccess(waitingQueue);
            public void waitForAccess(KThread thread) {
                  Lib.assertTrue(Machine.interrupt().disabled());
                  getThreadState(thread).waitForAccess(this);
            }
```

We need to implement two methods in ThreadState: acquire(...) and watForAccess(...)

```
[ThreadState]     public void acquire(PriorityQueue waitQueue) {
                        waitQueue.ower = Thread.currenThread()
                  }
[ThreadState] public void waitForAccess(PriorityQueue waitQueue) {
                  waitQueue.push(this);
                  current = ThreadState of currenThread
                  owner = the owner of the waitQueue
```

```
                add current -> owner.donorList
                // here is how we can make it
                // more efficient in order to lookup later
                ower.maxDonatePriority = max(owner.maxDonatePriority,
current.priority)
           }
```

[ThreadState]

```
public int getEffectivePriority() {
// since we already compute
// the max priority from other thread
return priority + this. maxDonatingPriority;
}
```

class PriorityQueue: we need to use priority Queue (built-in library for this class)

```
Queue<ThreadState> queue = new PriorityQueue<ThreadState>()

public KThread nextThread() {
      ThreadState next = queue.pop()
      return next.getThread()
}

public KThread pickNextThread() {
      ThreadState next = queue.peek()
      return next.getThread()
}
```

Task 6: Implement `Boat` class

The state of the **`Boat`** includes:

Set location `OAHU = 0`;
Set location `MOLOKAI = 1`;
A lock (`Lock lock`), which for locking the boat when someone has taken it.
A condition variable (`Condition condition`), which for whether someone should give up control of the boat
The number of children on Oahu, (`int childrenOnOahu`), which is initialized 0
The number of children on Molokai, (`int childrenOnMolokai`), which is initialized 0
The number of adults on Oahu, (`int adultsOnOahu`), which is initialized 0
The number of children on the boat, (`int childrenOnBoat`), which is initialized 0
The number of children last seen on Oahu, (`int lastReportedChildrenOnOahu`), which is initialized 0
The number of children last seen on Molokai, (`int lastReportedChildrenOnMolokai`), which is initialized 0
The number of adults last seen on Oahu, (`int lastReportedAdultsOnOahu`), which is initialized 0
The current location of the boat, (`int boatLocation`), which is initialized `OAHU`
The signal when everybody arrives to Molokai, (`boolean finished`), which is initialized `false`

First, we will create threads for children and adults in the `begin` method; this method will call `AdultItinerary` and `ChildItinerary` method to guide how threads should run to finish the job of moving people from Oahu to Molokai.

1. Implement the `AdultItinerary` method:

```
void Boat::AdulItinerary()
     adultsOnOahu++
     lock → Acquire()
     int currentLocation  ← OAHU
     while (!finished)
          if (The adult and the boat is on OAHU, the boat is
          empty,  and there is a child on MOLOKAI)
               adultOnOahu--;
               Count the number of children on OAHU to report to
               other island
               Adult rows to MOLOKAI
               Update boatLocation and currentLocation to
               MOLOKAI
               Report the number of children have counted on
               OAHU
               condition → Wake()
               condition → Sleep()
          else
               if everybody is on the MOLOKAI
                    finished ← true
               condition → Wake()
               condition → Sleep()
```

2. Implement the `ChildItinerary` method:

```
void Boat::ChildItinerary()
     childrenOnOahu++;
     lock → Acquire()
```

```
int currentLocation ← OAHU
while (!finished)
      if (The child and the boat is on OAHU and
      childrenOnOahu > 0)
           if ( childrenOnBoat == 0)
                 childrenOnOahu--
                 childrenOnBoat++
                 Find another child for pilot
                 Child ride to MOLOKAI
                 Update currentLocation and boatLocation to
                 MOLOKAI
                 childrenOnBoat--
                 childrenOnMolokai++
           else if ( childrenOnBoat == 1)
                 childrenOnOahu--
                 childrenOnBoard++
                 Count the number of adult and children on
                 OAHU
                 Child rows to MOLOKAI
                 Update currentLocation and boatLocation to
                 MOLOKAI
                 childrenOnBoat--
                 childrenOnMolokai++
                 Report the number of adults and children
                 have counted on OAHU
                 if ( nobody is on OAHU)
                      finished ← true
           condition → Wake()
           condition → Sleep()

      else if ( currentLocation and boatLocation is on
      MOLOKAI, and   there are people on OAHU )
           childrenOnMolokai--
           Count the number of children on MOLOKAI
           Child rows to OAHU
           Update currentLocation and boatLocation to OAHU
           childrenOnOahu++
           Report the number of children have counted on
           MOLOKAI
           condition → Wake()
           condition → Sleep()
      else
           condition → Wake()
           condition → Sleep()
```